Corrigé DM4

Partie 1

1. Si $i \in [\![1, n]\!]$, l'ensemble $E_{i,i}$ est par définition l'ensemble des non terminaux qui engendrent la lettre m_i. Comme G est sous forme normale de Chomsky, pour calculer $E_{i,i}$, il suffit de parcourir toutes les règles de G et pour toutes celles qui sont de la forme $X \to m_i$, ajouter X à l'ensemble $E_{i,i}$.

2. Soit $i, j \in [\![1, n]\!]$ tels que $i < j$. Montrons l'égalité d'ensembles demandée par double inclusion.

   Si il existe $k \in [\![i, j-1]\!]$ et $Y, Z \in V$ tels que $X \to YZ$, $Y \in E_{i,k}$ et $Z \in E_{k+1,j}$, alors par définition $Y \Longrightarrow^\star m_i...m_k$ et $Z \Longrightarrow^\star m_{k+1}...m_j$. Mézalors $X \Longrightarrow^\star m_i...m_k m_{k+1}...m_j$ et donc $X \in E_{i,j}$.

   Réciproquement, si $X \in E_{i,j}$ alors $X \Longrightarrow^\star m_i...m_j$. Comme $i < j$, le mot $m_i...m_j$ contient au moins deux lettres. Comme G est sous forme normale de Chomsky, on en déduit que la première dérivation de $m_i...m_j$ à partir de X est nécessairement de la forme $X \Longrightarrow YZ$ pour $Y, Z \in V$. Grâce au "lemme fondamental", on en déduit l'existence de $k \in [\![i-1, j]\!]$ tel que $Y \Longrightarrow^\star m_i...m_k$ et $Z \Longrightarrow m_{k+1}...m_j$. Mais on sait d'autre part que la grammaire G ne permet pas de produire le mot ε : l'entier k est donc compris entre i et $j-1$ ce qui conclut.

3. Pour le mot *abab*, on obtient la matrice suivante :

$$\begin{pmatrix} T & X,Z & X,T & S,X,Z \\ \emptyset & Y,Z & Y,T & X,Z \\ \emptyset & \emptyset & T & X,Z \\ \emptyset & \emptyset & \emptyset & Y,Z \end{pmatrix}$$

   On explicite le calcul du coefficient en position $(1,3)$. Pour le déterminer, il faut trouver tous les non terminaux qui engendrent un élément du produit cartésien des ensembles $E_{1,1}$ et $E_{2,3}$ (encadrés sur fond grisé ci-dessus), c'est-à-dire $\{TY, TT\}$. On en trouve un : X. A ceux-ci s'ajoutent tous les non terminaux qui engendrent un élément du produit cartésien de $E_{1,2}$ et $E_{3,3}$ (encadrés sur fond blanc), à savoir $\{XT, ZT\}$. Là encore, on en trouve un : T.

   Comme l'axiome S fait partie de l'ensemble $E_{1,n}$, on en déduit que *abab* est engendré par G_{ex}.

4. La première boucle pour calcule les ensembles $E_{i,i}$ d'après la question 1. La deuxième boucle pour calcule les ensembles $E_{i,j}$ pour $1 \leq i < j \leq n$ d'après la question 2 à condition que tous les $e_{i',j'}$ nécessaires au calcul de $e_{i,j}$ aient déjà été calculés lorsque le calcul de $e_{i,j}$ commence. Mais c'est bien le cas, car la question 2 indique que les ensembles dont on a besoin pour calculer $E_{i,j}$ sont ceux grisés sur le dessin suivant :

$$\begin{pmatrix} E_{1,1} & \cdots & E_{1,i} & \cdots & E_{1,j-1} & E_{1,j} & \cdots & E_{1,n} \\ \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ E_{i,1} & \cdots & E_{i,i} & \cdots & E_{i,j-1} & E_{i,j} & \cdots & E_{i,n} \\ E_{i+1,1} & \cdots & E_{i+1,i} & \cdots & E_{i+1,j-1} & E_{i+1,j} & \cdots & E_{i+1,n} \\ \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ E_{j,1} & \cdots & E_{j,i} & \cdots & E_{j,j-1} & E_{j,j} & \cdots & E_{j,n} \\ \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ E_{n,1} & \cdots & E_{n,i} & \cdots & E_{n,j-1} & E_{n,j} & \cdots & E_{n,n} \end{pmatrix}$$

Ainsi, il suffit d'avoir calculé tous les coefficients sur les surdigaonales (la surdiagonale numéro 0 correspondant à la diagonale elle-même) précédant la surdiagonale sur laquelle se trouve $e_{i,j}$ pour disposer de tous les ensembles nécessaires à la construction de cet ensemble : l'ordre de calcul des $e_{i,j}$ est adapté à la relation récurrente qui les lie.

En fin d'algorithme, le coefficient $e_{1,n}$ est égal à l'ensemble $E_{1,n}$. L'axiome S s'y trouve si et seulement si $S \Longrightarrow^\star m_1...m_n = m$ c'est-à-dire si et seulement si $m \in L(G)$.

5. On considère que déterminer le type d'une règle — soit de la forme $X \to a$, soit de la forme $X \to YZ$ —, extraire son membre droit dans le premier cas, extraire les deux lettres de son membre droit dans le second et extraire son membre gauche dans les deux cas se fait en temps constant.

   L'initialisation de la matrice E se fait en temps $O(|m|^2)$. Le calcul de la diagonale de la matrice E (première boucle pour) nécessite $O(|m||\mathcal{R}|)$ opérations où $|\mathcal{R}|$ est le nombre de règles dans la grammaire G en entrée. Le calcul des surdiagonales est grossièrement majoré par un $O(|m|^3|\mathcal{R}|)$.

   On en déduit que la complexité de l'algorithme de Cocke-Younger-Kasami est en $O(|m|^3|\mathcal{R}|)$.

   Remarque : La majoration grossière du calcul des surdiagonales ne l'est en fait pas tellement. Pour calculer la surdiagonale d, il faut calculer tous les $e_{i,i+d}$ pour $i \in [\![0, n-d]\!]$ (connaître i et d permet de déterminer la colonne du coefficient). Pour ce faire on considère tous les $k \in [\![i, d+i-1]\!]$ et pour chacun, on parcourt les règles de G. On obtient donc une complexité de l'ordre de

$$\sum_{d=1}^{n-1}\sum_{i=0}^{n-d}\sum_{k=i}^{d+i-1}|\mathcal{R}| = \sum_{d=1}^{n-1}\sum_{i=0}^{n-d} d|\mathcal{R}| = \sum_{d=1}^{n-1}(n-d+1)d|\mathcal{R}| = \Theta(n^3|\mathcal{R}|) \text{ où } n = |m|.$$

6. Si gex représente G_{ex}, gex.nb_variables vaut 5 et gex.nb_regles vaut 8. On numérote les non terminaux de G_{ex} comme suit (le numéro de S étant fixé) :

non terminal	S	T	X	Y	Z
numéro	0	1	2	3	4

   Les règles $S \to XY$ et $Y \to b$ sont représentées respectivement par r0 et r6.

```
regle r0 = {.type=2, .membre_gauche=0, .lettre='?', .variable1=2, .variable2=3};
regle r6 = {.type=1, .membre_gauche=3, .lettre='b', .variable1=-1, .variable2=-1};
```

7. Etant donné la façon de représenter un ensemble de non terminaux imposée par l'énoncé, on commence par déclarer une matrice de tableaux de booléens de taille $|m| \times |m|$. Chaque tableau est initialisé de sorte à ce qu'il représente un ensemble de non terminaux vide.

   On remplit ensuite correctement la diagonale de cette matrice : pour savoir si une règle est de la forme $X \to a$, il suffit d'observer son type. Puis on complète les surdiagonales : le seul point un peu délicat est de retrouver comment exprimer j en fonction de d et i lorsqu'on considère le coefficient $e_{i,j}$ sur la surdiagonale d mais ce travail a déjà été fait dans la remarque de la question 5.

   On n'oublie pas de libérer tout ce qui a été alloué sur le tas dans le bon ordre non sans avoir sauvegardé au préalable le résultat souhaité. Ce dernier consiste en la valeur du booléen situé dans la case 0 (puisque c'est le numéro de l'axiome par convention) du tableau situé en haut à droite de la matrice generateurs.

```
bool CYK(grammaire* g, char mot[])
{
    int nr = g->nb_regles;
    int nv = g->nb_variables;
```

```
    int longueur_mot = strlen(mot);
    bool*** generateurs = (bool***)malloc(sizeof(bool*)*longueur_mot);
    for (int i = 0; i < longueur_mot; i++)
    {
        generateurs[i] = (bool**)malloc(sizeof(bool**)*longueur_mot);
        for (int j = 0; j < longueur_mot; j++)
        {
            generateurs[i][j] = (bool*)malloc(sizeof(bool)*nv);
            for (int k = 0; k < nv; k++)
            {
                generateurs[i][j][k] = false;
            }
        }
    }
    //Initialisation de la diagonale
    for (int i = 0; i < longueur_mot; i++)
    {
        for (int j = 0; j < nr; j++)
        {
            regle r = g->productions[j];
            if (r.type == 1 && mot[i] == r.lettre)
            {
                int var = r.membre_gauche;
                generateurs[i][i][var] = true;
            }
        }
    }
    //Calcul des surdiagonales
    for (int d = 1; d < longueur_mot; d++)
    {
        for (int i = 0; i < longueur_mot - d; i++)
        {
            //Remplissage de la case (i,d+i) sur la diagonale d
            for (int k = i; k < d+i; k++)
            {
                for (int j = 0; j < nr; j++)
                {
                    regle r = g->productions[j];
                    //L'ordre de ces tests et la paresse du 'et' sont importants
                    if (r.type == 2 && generateurs[i][k][r.variable1] == true &&␣
↪generateurs[k+1][d+i][r.variable2])
                    {
                        generateurs[i][d+i][r.membre_gauche] = true;
                    }
                }
            }
        }
    }
```

```
    bool res = generateurs[0][longueur_mot-1][0];
    //Libérations
    for (int i = 0; i < longueur_mot; i++)
    {
        for (int j = 0; j < longueur_mot; j++)
        {
            free(generateurs[i][j]);
        }
        free(generateurs[i]);
    }
    free(generateurs);
    return res;
}
```

8. Pour obtenir ces réponses il sera sûrement nécessaire de se doter de fonctions permettant d'initialiser une grammaire, de libérer l'espace occupé sur le tas par une grammaire et d'ajouter une règle à une grammaire donnée. Une fois ceci fait, on peut vérifier que *abab* est bien un élément de $L(G)$ (on peut même modifier CYK de sorte à afficher la matrice qui y est contruite afin de vérifier qu'elle coïncide avec celle de la question 3) et on constate que m_2 est dans $L(G)$ mais pas m_1.

9. Si $\varepsilon \in L(G)$, la mise sous forme normale de Chomsky de G consiste à construire une grammaire sous forme normale de Chomsky engendrant $L(G) \setminus \{\varepsilon\}$ puis à y ajouter la règle $S \to \varepsilon$. On peut par exemple attribuer à cette règle spéciale le type 0 dans notre implémentation et modifier CYK ainsi :
   - Si le mot m en entrée est égal à ε, on parcourt les règles de G à la recherche de l'éventuelle règle de type 0 : si on la trouve, on renvoie true, sinon, on renvoie false.
   - Sinon, on supprime l'éventuelle règle de type 0 présente dans la grammaire G et on obtient ainsi une grammaire G' engendrant $L(G) \setminus \{\varepsilon\}$. On applique alors l'algorithme de la question 7 à G' (qui évidemment n'engendre par ε) et $m \neq \varepsilon$.

10. Have fun !

Remarque : En pratique, l'algorithme CYK n'est pas utilisé : il est trop lent. Il a néanmoins l'avantage de pouvoir s'appliquer à une grammaire arbitraire (puisqu'une grammaire quelconque peut toujours être mise sous forme normale de Chomsky, modulo ε dont le cas particulier peut être traité avec la question 9). La partie 2 présente un algorithme plus efficace mais qui ne s'applique pas à toutes les grammaires.

Partie 2

1. Le mot $m = \underbrace{\mathsf{sym}\ \mathsf{sym}\ ...\mathsf{sym}}_{n \text{ fois}}\#$ est engendré par la grammaire G via la dérivation gauche suivante :

$$\begin{aligned} S &\Longrightarrow L\# \\ &\Longrightarrow EL\# \\ &\Longrightarrow \mathsf{sym}L\# \\ &\Longrightarrow \mathsf{sym}\,EL\# \\ &\Longrightarrow \mathsf{sym}\,\mathsf{sym}L\# \\ &... \\ &\Longrightarrow \mathsf{sym}\,...\,\mathsf{sym}L\# \\ &\Longrightarrow m \end{aligned}$$

   Il remplit les deux contraintes imposé par l'énoncé.

2. Le principe de la fonction parseX est de "consommer" le non terminal X en parallèle de la suppression en tête de liste du plus long mot que X a pu générer. Les fonctions demandées s'implémentent ainsi :

```
let rec parseS l = match l with
  |(Sym|Lpar|Eof)::_ -> (match parseL l with
                          |Eof::q -> q
                          |_ -> raise SyntaxError)
  |[] |Rpar::_ -> raise SyntaxError
and parseL l = match l with
  |(Sym|Lpar)::_ -> parseL (parseE l)
  |(Rpar|Eof)::_ -> l
  |[] -> raise SyntaxError
and parseE l = match l with
  |Sym::q -> q
  |Lpar::q -> (match parseL q with
               |Rpar::r -> r
               |_ -> raise SyntaxError)
  |_ -> raise SyntaxError
```

   Donnons quelques explications pour parseE, par exemple :

   - Si on doit faire correspondre le non terminal E avec une liste commençant par sym, on applique la règle $E \to \mathsf{sym}$ ce qui permet de faire directement coïncider E et la tête de la liste : on en renvoie donc la queue pour poursuivre l'analyse.
   - Si on doit faire correspondre le non terminal E avec une liste commençant par (, on utilise la règle $E \to (L)$. Cette dernière introduit une parenthèse ouvrante qui coïncide avec la parenthèse ouvrante de début de liste : on les supprime toutes les deux et il reste à faire coïncider $L)$ avec la queue de la liste. On appelle donc parseL et après avoir consommé L il faudra nécessairement avoir une liste commençant par) pour avoir correspondance.
   - Dans les autres cas, on essaie de faire correspondre E soit à un mot vide, soit à un mot commençant par) soit à un mot commençant par # mais E ne permet pas de générer de tels mots et on obtient une erreur car E ne pourra pas être consommé lors de l'analyse.

3. On vérifie que tous les non terminaux dérivés au cours de l'analyse ont bien été consommés. Il faut également rattraper l'exception SyntaxError :

```
let accepts mot =
  try parseS mot = [] with SyntaxError -> false
```

4. Le symbole L est nul puisque $L \to \varepsilon$. Le symbole S n'est pas nul car tout mot dérivé de S contient le terminal #. De même, E n'est pas nul car tout mot dérivé de E contient (ou sym.

5. La valeur de $\textsc{Nul}(X)$ ne peut évoluer que de false vers true. Par conséquent, le nombre de false parmi les $|N|$ valeurs de $\textsc{Nul}(X)$ à calculer, qui est un entier naturel, décroît au fil des itérations. S'il ne change pas d'une étape à la suivante, l'algorithme termine immédiatement. Sinon, il décroît strictement et ceci ne peut arriver que $|N| < \infty$ fois.

6. On montre par récurrence sur le nombre d'étapes n de cette dérivation que si $X \Longrightarrow^\star \varepsilon$ alors $\textsc{Nul}(X) =$ true en fin d'algorithme. Si $X \Longrightarrow \varepsilon$ alors $X \to \varepsilon$ est une règle et l'étape 2 garantit que $\textsc{Nul}(X) =$ true. Si $X \Longrightarrow^{n+1} \varepsilon$ alors $X \Longrightarrow X_1 ... X_p$ avec pour tout i, $X_i \Longrightarrow^k \varepsilon$ avec $k \leq n$. Par hypothèse de récurrence on aura $\textsc{Nul}(X_i) =$ true et là encore l'étape 2 garantit le calcul correct de $\textsc{Nul}(X)$.

   Réciproquement, on montre que la propriété : "si $\textsc{Nul}(X) =$ true alors $X \Longrightarrow^\star \varepsilon$" est un variant pour la boucle tant que implicite à l'étape 2 de l'algorithme. Cette propriété est vraie avant d'entrer dans cette boucle pour la première fois puisqu'à cet instant tous les $\textsc{Nul}(X)$ sont égaux à false et la propagation est immédiate.

7. On obtient les 6 ensembles suivants :

$\textsc{Premiers}(S)$	$\textsc{Suivants}(S)$	$\textsc{Premiers}(L)$	$\textsc{Suivants}(L)$	$\textsc{Premiers}(E)$	$\textsc{Suivants}(E)$
{sym, (, #}	$\emptyset$	{sym, (}	{), #}	{sym, (}	{sym, (,), #}

8. Intuitivement, la grammaire G construit les listes par la gauche et G' par la droite. On montre l'égalité de $L(G)$ et $L(G')$ par double inclusion. On traite l'inclusion $L(G) \subset L(G')$, l'autre étant similaire.

   On montre d'abord par récurrence forte sur la longueur de cette dérivation que si $L \Longrightarrow^\star u$ avec $u \in T^+$ alors il existe v, w deux mots tels que $u = vw$, $L \Longrightarrow^\star v$ et $E \Longrightarrow^\star w$. Les plus courtes dérivations à partir de L menant à un mot de T^+ sont $L \Longrightarrow EL \Longrightarrow \text{sym}L \Longrightarrow$ sym et $L \Longrightarrow EL \Longrightarrow E \Longrightarrow$ sym. Dans les deux cas, on a sym $= \varepsilon$sym avec $L \Longrightarrow \varepsilon$ et $E \Longrightarrow^\star$ sym ce qui conlut l'initialisation.

   De plus, une telle dérivation de taille $n+1$ commence nécessairement par $L \Longrightarrow EL$ car $u \neq \varepsilon$. Elle se poursuit avec deux dérivations de longueur inférieure à n : $E \Longrightarrow^\star u_1$ et $L \Longrightarrow^\star u_2$ avec $u = u_1 u_2$ et on peut donc appliquer l'hypothèse de récurrence à la seconde : il existe u_3, w tel que $L \Longrightarrow^\star u_3$, $E \Longrightarrow w$ et $u_2 = u_3 w$. On a donc

$$L \Longrightarrow^\star \underbrace{u_1 u_3}_{=v} w$$

   avec v qui se dérive de EL donc de L et w qui se dérive de E comme attendu.

   On montre à présent que si $X \Longrightarrow^\star u$ alors $X' \Longrightarrow^\star u$. Cela montre en particulier que $L(G) \subset L(G')$. On procède par récurrence sur la longueur de la dérivation concernée. L'hérédité se divise en trois cas selon que X vaut S, E ou L. Le seul cas non immédiat est celui où $X = L$, traitons le :

   Si $L \Longrightarrow^\star u$ est une dérivation de longueur plus grande que deux, alors $u = vw$ avec $L \Longrightarrow^\star v$ et $E \Longrightarrow^\star w$ des dérivations strictement plus courtes d'après le lemme précédent. Par hypothèse de récurrence, $L' \Longrightarrow^\star v$ et $E' \Longrightarrow^\star w$ et ainsi $L' \Longrightarrow L'E' \Longrightarrow^\star vw = u$.

9. On commence par calculer les ensembles PREMIERS et SUIVANTS :

PREMIERS(S')	SUIVANTS(S')	PREMIERS(L')	SUIVANTS(L')	PREMIERS(E')	SUIVANTS(E')
{sym, (, #}	$\emptyset$	{sym, (}	{sym, (,), #}	{sym, (}	{sym, (,), #}

Par ailleurs, le seul non terminal nul est L'. On en déduit la table LL suivante :

	sym	(	)	#
S'	$L'\#$	$L'\#$		$L'\#$
L'	ε **OU** $E'L'$	ε **OU** $E'L'$	ε	ε
E'	sym	(L')		

Deux des cases contiennent deux choix possibles : on ne peut par exemple pas analyser un mot commençant par sym lorsque le non terminal à consommer est L' car on ne sait pas quelle règle utiliser. On ne peut donc pas implémenter un algorithme accepts en suivant les mêmes principes qu'aux questions 2 et 3 pour la grammaire G'.

Remarques culturelles :

- *La partie deux résout le problème du mot en analysant les mots de manière descendante : on reconstruit une dérivation pour le mot donné à partir de l'axiome. Il existe des méthodes d'analyse dites ascendantes qui tentent plutôt de "remonter" vers l'axiome en partant du mot à analyser.*
- *L'analyseur étudié ici est un analyseur dit LL (Left-to-right, Leftmost derivation) car il lit le mot à analyser de gauche à droite et en construit une dérivation gauche. Plus précisément, c'est un analyseur LL(1) car pour déterminer quelle règle appliquer lorsqu'on veut dériver un non terminal, on observe un seul symbole du mot à analyser. On peut étendre la notion d'annalyseur LL(1) à celle d'analyseur LL(k) pour lequel le choix de la règle à utiliser s'appuie sur les k premiers symboles du mot à analyser.*
- *Les grammaires pour lesquelles la table LL (au sens défini par l'énoncé) a des cases contenant au plus un symbole sont appelées grammaires LL(1). Pour ces dernières, un analyseur LL(1) permet de résoudre efficacement le problème du mot, mais malheureusement, toutes les grammaires ne sont pas équivalentes à une grammaire LL(1). On peut résumer ce sujet de DM ainsi : le premier algorithme traite toutes les grammaires mais est peu efficace ; le deuxième algorithme est plus efficace mais ne traite que certaines grammaires.*
- *On peut souligner la similarité entre une table LL(1) et la table des transitions d'un automate fini. Cette dernière n'est pas fortuite et inspire la réflexion suivante : une analyse LL(1) est possible lorsque la grammaire est "déterministe", autrement dit lorsqu'il n'y a qu'un seul choix possible de règle à utiliser dans une dérivation gauche pour le mot considéré étant donné sa première lettre.*